关于高级语言和OS

- **你应该用什么样的编程语言来实现操作系统内核？**

使用golang进行对比

C的优势

- C提供了大量的控制能力，可以C可以**完全控制内存分配和释放**
- C语言**几乎没有隐藏的代码**，你几乎可以在阅读C代码的时候想象到对应的RISC-V机器指令是什么
- 通过C可以有**直接内存访问能力**，你可以读写PTE的bit位或者是设备的寄存器
- 使用C会有**极少的依赖**，你**几乎可以直接在硬件上运行C程序。**你们可以在XV6启动过程中看到这一点， **只通过几行汇编代码，你就可以运行C代码**

但是

很难写出安全的C代码，存在各种各样的bug.

- buffer overrun，比如数组越界，撑爆了Stack等等。
- use-after-free bugs，你可能会释放一些仍然在使用的内存，之后其他人又修改了这部分内存。
- 而且当线程共享内存时，很难决定内存是否可以被释放。

（在2017年有40个Linux Bugs可以让攻击者完全接管机器。很明显，这些都是非常严重的Bugs，这些Bug是由buffer overrun和一些其他memory-safety bug引起。）

高级编程语言

- **memory-safety**（要么当它们发生时程序运行时会检查数组是否越界，如果越界了就panic；要么高级编程语言不允许你写出引起Bug的代码，所以这些问题完全不可能出现。）
- Type safety，类型安全
- 通过GC实现了自动的内存管理。
- 对并发更友好
- 有更好的抽象，接口和类等面向对象的语法使得你可以写出更加模块化的代码。

但是

- 高级编程语言通常有**更差的性能。**额外的代价（High Level Language Tax）
  - 比如说在索引一个数组元素时检查数据边界，比如说检查空指针，比如说类型转换。
  - 除此之外，GC也不是没有代价的，需要花费一些时间来跟踪哪些对象可以被释放。
- **高级编程语言与内核编程本身不兼容。**
  - 高级编程语言没有直接访问内存的能力，因为这从原则上违反了Type safety。
  - 高级编程语言不能集成汇编语言，而在内核中的一些场景你总是需要一些汇编程序，比如说两个线程的context switching，或者系统启动
  - 编程语言**本身支持的并发与内核需要的并发并不一致**，比如我们在调度线程的时候，一个线程会将锁传递给另一个线程。一些并发管理模式在用户程序中不太常见，但是在内核中会出现。

解决heap耗尽的问题

系统调用之前调用reserve来预留足够的内存，如果没有就等待。（等待的时候没有持有锁，也没有任何资源）

优点

- 在内核中没有检查。你不需要检查内存分配是否会失败，在我们的例子中这尤其得好，因为在Golang中内存分配不可能会失败。
- 这里没有error handling代码。
- 这里没有死锁的可能，因为你在最开始还没有持有锁的时候，就避免了程序继续执行。

如何知道自己要保留多少的内存？

调包进行代码的静态分析。

- 我们对于内核文本采用了大页，以避免TLB的代价。
- 我们有针对每个CPU的网卡队列，这样可以避免CPU核之间同步。
- 我们有RCU实现了不需要读锁的Directory Cache。

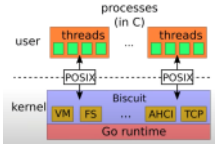
why? Golang

- 这是一个静态编译的编程语言，基本上来说这是一种高性能编程语言。
- Golang被设计成适合系统编程，而内核就是一种系统编程所以Golang也符合这里的场景。例如：
  - Golang非常容易调用汇编代码，或者其他的外部代码
  - Golang能很好地支持并发
  - Golang非常的灵活
- Golang带有Garbage Collector。使用高级编程语言的一个优点就是你不需要管理内存，而GC是内存管理的核心。

(Rust相比Golang还有一个缺点，Rust认为高性能程序不能有GC，所以Rust不带GC。实际上Rust的类型系统以一种非常聪明且有趣的方式实现，所以GC对于Rust并不是必须的。这里涉及到一个有趣的问题：通过高级编程语言实现内核时，GC的代价到底有多少？而Rust通过不使用GC而跳过了这个问题。)

学生提问：如果我们这里使用Rust而不是Golang来实现高级编程语言内核，通过一定的优化有没有可能达到比C内核更高的性能？

Frans教授：因为我们没有做过这样的实验，所以我就猜一下。我觉得不会有比C内核更高的性能，但是基本在同一个范围内。因为C是如此的底层，你可以假设你在Rust做的工作，都可以在C中完成。



- 每个用户程序都有属于自己的Page Table。
- 用户空间和内核空间的内存是由硬件隔离的，也就是通过PTE的User/Kernel bit来区分。
- 每个用户线程都有一个对应的内核线程，这样当用户线程执行系统调用时，程序会在对应的内核线程上运行。如果系统调用阻塞了，那么同一个用户地址空间的另一个线程会被内核调度起来。
- 如之前提到的，内核线程是由Go runtime提供的goroutine实现的。如果你曾经用Golang写过用户空间程序，其中你使用go关键字创建了一个goroutine，这个goroutine就是Biscuit内核用来实现内核线程的goroutine。

学生提问：我认为Golang更希望你使用channel而不是锁，所以这里在实现的时候会通过channel取代之前需要锁的场景吗？

Frans教授：这是个好问题，我会稍后看这个问题，接下来我们有几页PPT会介绍我们在Biscuit中使用了Golang的什么特性，但是我们并没有使用太多的channel，大部分时候我们用的就是锁和conditional variable。所以某种程度上来说Biscuit与XV6的代码很像，而并没有使用channel。**我们在文件系统中尝试过使用channel，但是结果并不好，相应的性能很差，**所以我们切换回与XV6或者Linux类似的同步机制。

这里的解决方法是使用了高级编程语言的特性。Golang实际上非常容易做静态分析，Go runtime和Go生态里面有很多包可以用来分析代码，我们使用这些包来计算系统调用所需要的内存。所以你可以想象，如果你有一个read系统调用，我们可以通过系统调用的函数调用图查看比如函数f调用函数g调用函数h等等等等。我们可以做的是弄清楚这里调用的最大深度，对于最大的深度，计算这里每个函数需要的内存是多少。

比如说函数f调用了new，因为这是一个高级编程语言，我们知道new的对象类型，所以我们可以计算对象的大小。我们将所有的new所需要的内存加起来，得到了一个总和S，这就是这个调用图（或者说系统调用）任何时间可能需要的最大内存。

实际中并没有这么简单，会有点棘手。因为函数h可能会申请了一些内存，然后再回传给函数g。所以当h返回时，g会得到h申请的一些内存。这被称为escaping，内存从h函数escape到了函数g。

Frans教授：是的。这里的工具会计算最深函数调用时最大可能使用的内存量。所以它会计算出每个系统调用可能使用的最多内存，虽然实际中系统调用可能只会使用少的多的内存。但是保险起见，我们会为最坏情况做准备。一些系统调用内的for循环依赖于传给系统调用的参数，所以你不能静态地分析出内存边界是什么。所以在一些场景下，我们会标注代码并规定好这是这个循环最大循环次数，并根据这个数字计算内存总量S。

类似的，如果有你有递归调用的函数，谁知道会递归多少次呢？或许也取决于一个动态变量或者系统调用的参数。实际中，我们在Biscuit中做了特殊处理以避免递归函数调用。所以最后，我们才可能完成这里的内存分析。

学生提问：这里的静态内存分析工具，如果不是用来构建内核，它们通常会用来干嘛？

Frans教授：Go编译器内部使用它来完成各种各样的优化，并分析得出最优的编译方式。这里正好编译器使用了一个包，我们也可以使用同样的包。在后面你还可以看到，我们还将它用于一些其他特性，有这么一个包非常的方便

- 首先，我们需要让Go runtime运行在裸机之上。我们希望对于runtime不做任何修改或者尽可能少的修改，这样当Go发布了新的runtime，我们就可以直接使用。在我们开发Biscuit这几年，我们升级了Go runtime好几次，所以Go runtime直接运行在裸机之上是件好事。并且实际上也没有非常困难。Golang的设计都非常小心的不去依赖操作系统，因为Golang想要运行在多个操作系统之上，所以它并没有依赖太多的操作系统特性，我们只需要仿真所需要的特性。大部分这里的特性是为了让Go runtime能够运行起来，一旦启动之后，就不太需要这些特性了。
- 我们需要安排goroutine去运行不同的应用程序。通常在Go程序中，只有一个应用程序，而这里我们要用goroutine去运行不同的用户应用程序，这些不同的用户应用程序需要使用不同的Page Table。这里困难的点在于，Biscuit并不控制调度器，因为我们使用的是未经修改过的Go runtime，我们使用的是Go runtime调度器，所以在调度器中我们没法切换Page Table。Biscuit采用与XV6类似的方式，它会在内核空间和用户空间之间切换时更新Page Table。所以当进入和退出内核时，我们会切换Page Table。这意味着像XV6一样，当你需要在用户空间和内核空间之间拷贝数据时，你需要使用copy-in和copy-out函数，这个函数在XV6中也有，它们基本上就是通过软件完成Page Table的翻译工作。
- 另一个挑战就是设备驱动，Golang通常运行在用户空间，所以它并不能从硬件收到中断。但是现在我们在裸机上使用它，所以它现在会收到中断，比如说定时器中断，网卡中断，磁盘驱动中断等等，我们需要处理这些中断。然而在Golang里面并没有一个概念说是在持有锁的时候关闭中断，因为中断并不会出现在应用程序中，所以我们在实现设备驱动的时候要稍微小心。我们采取的措施是在设备驱动中不做任何事情，我们不会考虑锁，我们不会分配任何内存，我们唯一做的事情是向一个非中断程序发送一个标志，之后唤醒一个goroutine来处理中断。在那个goroutine中，你可以使用各种各样想要的Golang特性，因为它并没有运行在中断的context中，它只是运行在一个普通goroutine的context中。
- 前三个挑战我们完全预料到了，我们知道在创造Biscuit的时候需要处理它们，而最难的一个挑战却不在我们的预料之中。这就是heap耗尽的问题。所以接下来我将讨论一下heap耗尽问题，它是什么，它怎么发生的，以及我们怎么解决的？

对于堆耗尽的情况。

- 第一种方法我们在XV6中见过。如果XV6不能找到一个空闲的block cache来保存disk block，它会直接panic。这明显不是一个理想的解决方案。这并不是一个实际的解决方案，所以我们称之为strawman。
- 另一个strawman方法是，当你在申请一块新的内存时，你会调用alloc或者new来分配内存，你实际上可以在内存分配器中进行等待。这实际上也不是一个好的方案，原因是你可能会有死锁。假设内核有把大锁，当你调用malloc，因为没有空闲内存你会在内存分配器中等待，那么这时其他进程都不能运行了。因为当下一个进程想要释放一些内存时，但是因为死锁也不能释放。对于内核中有大锁的情况，这里明显有问题，但是即使你的锁很小，也很容易陷入到这种情况：在内存分配器中等待的进程持有了其他进程需要释放内存的锁，这就会导致死锁的问题。
- 下一个strawman方法是，如果没有内存了就返回空指针，你检查如果是空指针就直接失败，这被称为bail out。但是bail out并不是那么直观，进程或许已经申请了一些内存，那么你需要删除它们，你或许做了一部分磁盘操作，比如说你在一个多步的文件系统操作中间，你只做了其中的一部分，你需要回退。所以实际中非常难做对。